On the Node-Averaged Complexity of Locally Checkable Problems on Trees

Alkida Balliu
Gran Sasso Science Institute, L'Aquila, Italy

Sebastian Brandt
Helmholtz Center for Information Security, Saarbrücken, Germany

Fabian Kuhn
University of Freiburg, Germany

Dennis Olivetti
Gran Sasso Science Institute, L'Aquila, Italy

Gustav Schmid
University of Freiburg, Germany

Abstract

Over the past decade, a long line of research has investigated the distributed complexity landscape of locally checkable labeling (LCL) problems on bounded-degree graphs, culminating in an almost-complete classification on general graphs and a complete classification on trees. The latter states that, on bounded-degree trees, any LCL problem has deterministic *worst-case* time complexity $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, or $\Theta(n^{1/k})$ for some positive integer k, and all of those complexity classes are nonempty. Moreover, randomness helps only for (some) problems with deterministic worst-case complexity $\Theta(\log n)$, and if randomness helps (asymptotically), then it helps exponentially.

In this work, we study how many distributed rounds are needed *on average per node* in order to solve an LCL problem on trees. We obtain a partial classification of the deterministic *node-averaged* complexity landscape for LCL problems. As our main result, we show that every problem with worst-case round complexity $O(\log n)$ has deterministic node-averaged complexity $O(\log^* n)$. We further establish bounds on the node-averaged complexity of problems with worst-case complexity $\Theta(n^{1/k})$: we show that all these problems have node-averaged complexity $\widetilde{\Omega}(n^{1/(2^k-1)})$, and that this lower bound is tight for some problems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases distributed graph algorithms, locally checkable labelings, node-averaged complexity, trees

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.7

Related Version *Full Version*: `https://arxiv.org/abs/2308.04251` [1]

Funding This work has been partially funded by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY – CUP: D13C21000430001, and by the German Research Foundation (DFG), Grant 491819048.

1 Introduction

The family of locally checkable labeling (LCL) problems was introduced in the seminal work of Naor and Stockmeyer [22] and since then, understanding the distributed complexity of computing LCLs has been at the core of the research on distributed graph algorithms. Roughly speaking, LCLs are labelings of the nodes or edges of a graph $G = (V, E)$ with labels from a finite alphabet such that some local, constant-radius condition holds at all the nodes. In the distributed context, G represents a network and one typically assumes that the nodes

of G can communicate over the edges of G in synchronous rounds. If this communication is unrestricted, this is known as the LOCAL model of computation and if messages must consist of $O(\log n)$ bits (where n is the number of nodes), it is known as the CONGEST model. In our paper, we focus on the LOCAL model and we therefore do not explicitly analyze the required message sizes of our algorithms. We however believe that all our algorithms can be made to work in the CONGEST model with minor modifications.

Often LCL problems are studied in the context of bounded-degree graphs. In this case, LCLs include problems such as properly coloring the nodes of G with $\Delta + 1$ colors, where Δ is the maximum degree of G. By now, researchers have obtained a thorough understanding of the complexity landscape of distributed LCL problems in general bounded-degree graphs [13, 12, 16, 23, 6, 2] and also in more special graph families such as in particular in bounded-degree trees [17, 7, 12, 13, 3, 10]. Most of this work focuses on the classic notion of worst-case complexity: If all nodes start a computation at time 0 and communicate in synchronous rounds, how many rounds are needed until *all nodes* have decided about their outputs. In some case however, the worst-case round complexity might be determined by a small number of nodes that require a lot of time to compute their outputs, while most of the nodes find their outputs much faster. Consider for example the simple randomized $(\Delta+1)$-coloring algorithm where in every step, every node picks a random available color and permanently keeps this color if there is no conflict. It is not hard to show that in every step, every uncolored node becomes colored with constant probability [18]. Hence, while we need $\Omega(\log n)$ steps (and thus also $\Omega(\log n)$ rounds) until all nodes are colored, for each individual node, the expected time to become colored is constant and consequently the time that nodes need on average to become colored is also constant w.h.p. In some contexts (e.g., when considering the energy cost of a distributed algorithm), this average completion time per node is more meaningful than the worst-case completion time and consequently, researchers have recently showed interest in determining the *node-averaged* time complexity of distributed graph algorithms [15, 9, 14, 5]. In the present paper, we continue this work and we study the *node-averaged complexity of LCL problems in bounded-degree trees*. Before describing our contributions, we first briefly summarize some of the relevant previous work.

Previous results on node-averaged complexity. The first paper that explicitly considered the node-averaged complexity of distributed graph algorithms is by Feuilloley [15]. The paper mainly considers LCL problems on paths and cycles (i.e., on graphs of maximum degree 2). It is known that on paths and cycles, when considering the worst-case complexity of LCL problems, randomization does not help and the only complexities that exist are $O(1)$, $\Theta(\log^* n)$, and $\Theta(n)$ [22, 12, 13]. In [15], it is shown that for deterministic algorithms, the worst-case complexity and the node-averaged complexity of LCL problems on paths and cycles is the same. This for example implies that the classic $\Omega(\log^* n)$ lower bound of [20] for coloring cycles with a constant number of colors also applies to node-averaged complexity. While this is true for deterministic algorithms, it is also shown in [15] that the randomized node-averaged complexity of 3-coloring paths and cycles is constant. As sketched above and also explicitly proven in [9], the same is true for the more general problem of computing a $(\Delta + 1)$-coloring in arbitrary graphs. While the results of [15] imply results for general LCLs on paths and cycles, the additional work on node-averaged complexity focused on the complexity of specific graph problems, in particular on the complexity of well-studied classic problems such as computing a maximal independent set (MIS) or a vertex coloring of the given graph. Barenboim and Tzur [9] show that in graphs of small arboricity, some coloring problems have a deterministic node-averaged complexity that is significantly smaller than

the corresponding worst-case complexity. For example, it is shown that if the arboricity is constant, an $O(k)$-vertex coloring can be computed in node-averaged complexity $O(\log^{(k)} n)$ for any fixed integer $k \geq 1$, where $O(\log^{(k)} n)$ denotes the k times iterated logarithm of n. As one of the main results of [5], it was shown that the MIS lower bound of [19] can be generalized to show that even with randomization, computing an MIS on general (unbounded degree) graphs requires node-averaged complexity $\Omega(\sqrt{\log n / \log\log n})$. Hence, while the problem of coloring with $(\Delta + 1)$ colors and, as also shown in [5], the problem of computing a 2-ruling set have randomized algorithms with constant node-averaged complexity, the same thing is not true for the problem of computing an MIS.

LCL complexity in bounded-degree trees. One of the goals of this paper is to make a step beyond understanding individual problems and to start studying the landscape of possible node-averaged complexities of general LCL problems. We do this by studying LCL problems on bounded-degree trees, a graph family that we believe is relevant and that has recently been studied intensively from a worst-case complexity point of view (e.g., [12, 13, 11, 7, 16, 23, 17]). In bounded-degree trees, for deterministic algorithms, exactly the following worst-case complexities are possible: $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, and $\Theta(n^{1/k})$ for some fixed integer $k \geq 1$. It was shown in [17] (and earlier for a special subclass of LCLs in [7] and for paths in [22, 13]) that on bounded-degree trees, there are no deterministic or randomized optimal algorithms with a time complexity in the range $\omega(1)$ to $o(\log^* n)$. Further, in [12], it was shown that even for general bounded-degree graphs, there are no deterministic LCL complexities in the range $\omega(\log^* n)$ to $o(\log n)$. Finally, it was shown in [13] that every LCL problem that requires $\omega(\log n)$ rounds on bounded-degree trees has a worst-case deterministic and randomized complexity of the form $\Theta(n^{1/k})$ for some fixed integer $k \geq 1$ (and all those complexities also exist). It is further known that randomization can only help for LCL problems with a deterministic complexity of $\Theta(\log n)$. Those problems have a randomized complexity of either $\Theta(\log n)$ or $\Theta(\log\log n)$ (and both cases exist) [13, 11].

1.1 Our Contributions

As our main result, we show that the $\Theta(\log n)$ complexity class vanishes when considering the node-averaged complexity of LCLs on bounded-degree trees.

▶ **Theorem 1.** *Let Π be an LCL problem for which there is an $O(\log n)$-round deterministic algorithm on bounded-degree trees. Then, Π can be solved deterministically with node-averaged complexity $O(\log^* n)$ on bounded-degree trees.*

A standard example for an LCL problem that requires $\Theta(\log n)$ rounds deterministically is the problem of 3-coloring a tree. So for 3-coloring Theorem 1 states that there is a deterministic distributed 3-coloring algorithm, for bounded degree trees, with node-averaged complexity $O(\log^* n)$ rounds. Meaning that the average node terminates after $O(\log^* n)$ rounds. Note that for 3-coloring trees deterministically, this is tight. As shown in [15], 3-coloring has deterministic node-averaged complexity $\Omega(\log^* n)$ even on paths. Below, we will use the 3-coloring problem as a simple example to illustrate some of the challenges in obtaining the above theorem, but first we state the rest of our results.

In addition to Theorem 1, we also investigate the node-averaged complexity of LCL problems that require polynomial time in the worst case (i.e., time $\Theta(n^{1/k})$ for some integer $k \geq 1$). We show that for such problems, also the node-averaged complexity is polynomial. However at least in some cases, it is possible to obtain a node-averaged complexity that is

significantly below the worst-case complexity. In [13], the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is defined as an example problem with worst-case complexity $\Theta(n^{1/k})$. We show that the node-averaged complexity of this LCL problem is significantly smaller.

▶ **Theorem 2.** *The deterministic node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$.*

Finally, we show that for a problem with worst-case complexity $\Theta(n^{1/k})$, this is essentially the best possible node-averaged complexity. Meaning that we also prove that our algorithm for hierarchical $2\frac{1}{2}$-coloring problems is optimal up to one $\log n$ factor.

▶ **Theorem 3.** *Let Π be an LCL problem with (deterministic or randomized) worst-case complexity $\Omega(n^{1/k})$. Then, the randomized node-averaged complexity of Π is $\Omega(n^{1/(2^k-1)}/\log n)$.*

Note that the algorithm of Theorem 2 is deterministic, and that the lower bound of Theorem 3 holds for randomized algorithms as well.

1.2 High-level Ideas and Challenges

We next discuss some of the ideas that lead to the known results about solving LCL problems on bounded-degree trees and we highlight some of the challenges that one has to overcome and some of the ideas we use to prove Theorems 1–3.

Rake-and-compress decomposition. We start by sketching a generic algorithm that can be used to solve all LCL problems in bounded-degree trees. The generic algorithm can be used to obtain algorithms with an asymptotically optimal worst-case complexity for all problems with worst-case complexity $\Omega(\log n)$. As a first step, the algorithm uses a technique that is known as *rake-and-compress* [21] to partition the nodes of a given tree $T = (V, E)$ into $O(\log n)$ layers such that each layer is either a rake layer that consists of a set of independent nodes or it is a compress layer that consists of a sufficiently separated set of paths. Every node in a rake layer has at most one neighbor in a higher layer, and in each path of a compress layer, the two nodes at the end have exactly one neighbor in a higher layer and the other nodes on the path have no neighbors in a higher layer.[1] Such a decomposition can be computed in an iterative process that produces the layers in increasing order. Given some tree (or forest), a rake layer can be obtained by taking the set of all leaf nodes[2] and a compress layer can be created by the paths (or more precicely by the inner part of the paths) induced by degree-2 nodes. It is not hard to show that when alternating rake and compress layers, this process completes after creating $O(\log n)$ layers [21].

Applying the decomposition. As an example of how to use rake-and-compress to solve an LCL problem, we look at the case of 3-coloring the nodes of a tree T. Given a decomposition into rake and compress layers, this can be done in $O(\log n)$ rounds. First, color each of the paths of the compress layers with $O(1)$ colors. This requires $O(\log^* n)$ rounds. Then, the 3-coloring of T is computed by starting at the highest layer of the decomposition. When processing a rake layer, each node is colored with a color different from its (at most one) neighbor in a higher layer. When processing a compress layer, we just have to 3-color the

[1] The actual decomposition that we use is a bit more complicated and the formal definition (see Definition 4) requires some additional details.

[2] When two degree-1 nodes are neigbors, one just takes one of the two nodes.

paths of the layer such that each node at the end of a path picks a color that differs from the color of its neighbor in a higher layer. Given the initial $O(1)$-coloring of the path, this can be done in constant time for each path. The time to compute the coloring is therefore proportional to the number of layers and thus $O(\log n)$. The generic algorithm for solving more general LCL problems is more involved, but still similar at a high level. While creating the decomposition, for each node v, one can create a list of labels that can be assigned to v such that the labeling of lower layer nodes that depend on v can still be completed. The LCL problem needs to allow labelings that are flexible enough such that when having long paths of nodes that each can be the root of an arbitrary subtree, the nodes of the path can still be labeled efficiently (in constant time given an appropriate initial coloring of the path).

Implementation with low node-averaged complexity. The main challenge to achieve node-averaged complexity $o(\log n)$ is the following. The generic algorithm first computes the decomposition and it then computes the labeling by starting with the nodes in the highest layers. In the worst case, we thus need $\Theta(\log n)$ rounds before even the label of a single node is determined. Moreover, most of the nodes are in the lowest layers, which are labeled at the very end of the algorithm. We therefore need to label most of the nodes already in the "bottom-up" phase when creating the rake and compress layers. For some problems, this is challenging: for example, in the 3-coloring problem, if we ever obtain a node with 3 neighbors of lower layers that have 3 different colors, then we cannot complete the solution in any valid way. Hence, we have to label the nodes in such a way that the "top-down" phase is still able to extend the partial labeling to a valid labeling of all the nodes. In the following high-level discussion, for simplicity, we assume that the tree has diameter $O(\log n)$ so that it suffices to create rake layers and we do not need compress layers. We further only look at the problem of 3-coloring T. This problem is significantly easier to handle than the general case. The solution for 3-coloring however already requires some of the ideas of the general case.

Let us therefore assume that we have an $O(\log n)$-diameter $O(1)$-degree tree T. If we only construct rake layers, we obtain $O(\log n)$ layers, where each layer is an independent set and except for a single node u in the top layer, every node has exactly one neighbor in a higher layer. We refer to u as the root and for each other node, we refer to the single neighbor in a higher layer as the parent. Note that when assigning a color to a node v in the top-down phase, only v's parent has already been assigned a color. To complete the top-down phase, it therefore suffices if every node v can choose its color from an arbitrary subset S_v of size 2 of the colors. Hence, if we try to color some nodes already in the bottom-up phase, we have to make sure that all the uncolored nodes still have at least two available colors. This is for example guaranteed as long as every uncolored node has at most one colored neighbor.

When constructing the layering we therefore proceed as follows. We only color nodes that have already been assigned to some rake layer. Whenever we decide to color a node v in the bottom-up phase, we also directly color the whole subtree of v.[3] The high-level idea of the algorithm to achieve this is as follows. After each rake step, i.e., after each creation of a new layer, we check whether or not there are some nodes that can be colored. Consider the situation after the t^{th} rake step, let $G^{(t)}$ be the set of nodes that have not been raked at that time (i.e., that have not been assigned to some layer), and let $R^{(t)}$ be the set of nodes that have already been assigned to some layer. Note that if a node $u \in G^{(t)}$ has some neighbor $v \in R^{(t)}$, then u will in the end be the unique neighbor of v in a higher layer. We

[3] After coloring the root of a subtree, the coloring of the subtree can be done in parallel while proceedings with the rest of the algorithm.

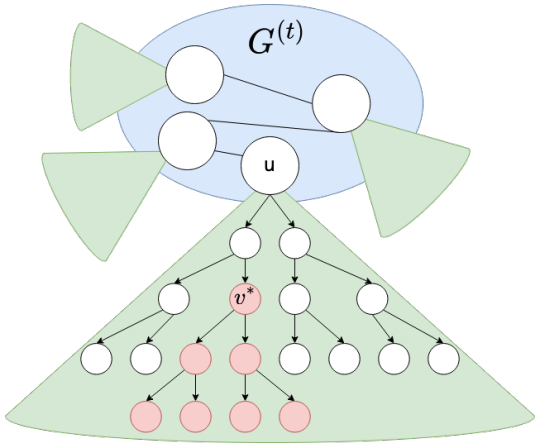

Figure 1 The graph $G^{(t)}$ of nodes that are not yet raked away is colored blue. The already raked away nodes $R^{(t)}$ are colored green. The node u chooses v^* since it has the largest subtree, colored in red, attached and both v^* as well as its entire subtree become colored.

can therefore think of the nodes in $G^{(t)}$ as the roots of the already raked subtrees. This is illustrated in Figure 1. After each rake step t, each node $u \in G^{(t)}$ tries to color some node at distance 2 in its subtree.[4] Node u chooses v^* to be a node at distance 2 in its subtree such that the subtree rooted at v^* has the largest number of uncolored nodes among all nodes at distance 2 of u in the subtree of u (observe that nodes can keep track of such numbers). If there are no colored 2-hop neigbors of v^* outside the subtree of v^* (i.e., no colored siblings of v^*), then u decides to color v^* and its complete subtree. Otherwise, no new nodes in u's subtree are getting colored. If v^* and its subtree get colored, then a constant fraction of the uncolored nodes in u's subtree get colored. Otherwise, a sibling v' of v^* with a larger subtree has already been colored while u was the root of the tree. Note that at this time, the subtree of v^* was already in the same state and therefore v' colored more nodes than v^* does. One can use this to show that whenever the height of a raked subtree increases, a constant fraction of the uncolored nodes gets colored. One can further show that this suffices to show that over the whole tree, a constant fraction of the remaining nodes gets colored every constant number of rounds and thus the node-averaged complexity is constant. The algorithm and the analysis for the general family of LCLs for which Theorem 1 holds uses similar basic ideas, dealing with the general case is however significantly more involved.

Improved upper bounds in the polynomial regime. We prove that the node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$. In order to give some intuition for this, we focus on the case $k = 2$ where the worst-case complexity is $\Theta(\sqrt{n})$. Instead of providing a formal definition of the problem, it is helpful to present the problem by describing how a worst-case instance for the problem looks like, and how a solution in such an instance looks like. A worst-case instance for this problem consists of a path P of length $\Theta(\sqrt{n})$, where to each node v_j of P is attached a path Q_j of length $\Theta(\sqrt{n})$. We call the nodes of the path P *p-nodes* and we call the nodes of a path Q_j *q-nodes*. For each path Q_j, the algorithm has to decide to either 2-color it or to mark the whole path as

[4] By only coloring nodes at distance at least 2 from u, we make sure that neighbors of nodes that are not yet layered remain uncolored.

decline. Then, the subpaths of P induced by nodes that are neighbors of q-nodes that output decline need to be labeled with a proper 2-coloring. In particular, *decline* is not allowed on p-nodes. Let us now describe an algorithm with optimal worst-case complexity for instances with a similar structure, but where the paths may have different lengths. For q-nodes, the algorithm first checks if the length of the path containing those nodes is $O(\sqrt{n})$ (note that, in order to perform this operation, the algorithm needs to know n, and it is actually unknown whether an LCL problem can have $\Theta(\sqrt{n})$ worst-case complexity when n is unknown). In such a case, the algorithm is able to produce a proper 2-coloring of the path. Otherwise, the path is marked as decline. Then, it is possible to prove that the subpaths of P induced by nodes having q-node neighbors that output decline must be of length $O(\sqrt{n})$, and hence they can be properly 2-colored in $O(\sqrt{n})$ rounds. We observe that in the worst-case instance described above, the majority of the nodes of the graph are q-nodes, and hence, from an average point of view, it would be fine if p-nodes spend more time. In fact, it is possible to improve the node-averaged complexity of the described algorithm by letting q-nodes run for at most $O(n^{1/3})$ rounds and p-nodes for at most $O(n^{2/3})$ rounds. In this case, a worst-case instance contains a path P of length $O(n^{2/3})$ and all paths Q_j are of length $O(n^{1/3})$. We obtain that both the p-nodes and the q-nodes contribute $O(n^{4/3})$ to the sum of the running times, obtaining a node-averaged complexity of $O(n^{1/3})$.

Lower bounds in the polynomial regime. It is known by [10] that if an LCL problem Π has worst-case complexity $o(n^{1/k})$, then it can actually be solved in $O(n^{1/(k+1)})$ rounds. The intuition about what determines the exact value of k in the complexity of a problem is related to how many compress layers of a rake-and-compress decomposition one can handle. In the example presented above, namely 3-coloring, one can handle an arbitrary number of compress paths and that is the reason why the problem can be solved in $O(\log n)$ rounds. In particular, no matter how many rake or compress operations have been applied, we can handle any compress path by producing a 3-coloring on it and leaving the endpoints uncolored (such nodes can decide their color after their higher layer neighbors picked a color), and this can be done fast. Not all problems are of this form, that is, for some problems we cannot handle an arbitrary amount of compress paths: it is possible to define problems in which different labels need to be used in compress paths of different layers (hierarchical $2\frac{1}{2}$ coloring is indeed such a problem where in fact p-nodes are not allowed to output *decline*). For such problems, it may not be possible at all to efficiently produce a valid labeling for long compress paths of layers that are too high, say of layers strictly more than k. In order to solve this issue, we can modify the generic algorithm sketched above by increasing the number of rake operations that are performed between each pair of compress operations. When using $\Omega(n^{1/(k+1)})$ rake operations at the beginning and between any two compress operations, the total number of compress layers is at most k. This however makes the algorithm slower, resulting in a complexity of $\Theta(n^{1/(k+1)})$ (while 3-coloring has worst-case complexity $\Theta(\log n)$).

In other words, for some LCL problems, compress paths are something that is difficult to handle, and the number of compress layers that we can recursively handle is what determines the complexity of a problem. If we can handle an arbitrary amount of compress layers, then the problem can be solved in $O(\log n)$ rounds, but if we can handle only a constant amount of compress layers, say k, then the complexity of the problem is $\Theta(n^{1/(k+1)})$. In [10] it is proved that, if a problem has complexity $o(n^{1/k})$, then it is possible to handle k compress layers, implying a complexity of $O(n^{1/(k+1)})$. We show that the same can be obtained by starting from an algorithm $\mathcal{A}$ with node-averaged complexity $o(n^{1/(2^k-1)}/\log n)$, implying that if a problem has complexity $\Omega(n^{1/k})$, then it cannot have node-averaged complexity

$o(n^{1/(2^k-1)}/\log n)$, since otherwise it would imply that the problem can actually be solved in $O(n^{1/(k+1)})$ rounds in the worst case, which then leads to a contradiction. Starting from an algorithm that only has guarantees on its node-averaged complexity instead of on its worst-case complexity introduces many additional challenges that we need to tackle. For example, in [10] it is argued that an $o(n^{1/k})$-rounds algorithm can never see both the endpoints of a carefully crafted path that is too long. This kind of reasoning, that is very common when we deal with worst-case complexity, does not work for node-averaged complexity.

Road Map. Section 2 provides some important definitions and a high-level description of the generic algorithm to solve LCLs on bounded-degree trees with optimal worst-case complexity. In Section 3, we present an algorithm with node-averaged complexity $O(\log^* n)$ that is able to solve all problems that have $O(\log n)$ worst-case complexity. The algorithm is based on the one discussed in Section 2.1, but we need to tackle several challenges to improve its node-averaged complexity. The proofs of Theorems 2 and 3 are in Appendices A and B.

2 Preliminaries

Node-averaged complexity. We start by defining the notion of node-averaged complexity as in [5]. Let $\mathcal{A}$ be an algorithm that solves a problem Π. Assume $\mathcal{A}$ is run on a given graph $G = (V, E)$. Let $v \in V$. We define $T_v^G(\mathcal{A})$ to be the number of rounds after which v terminates when running $\mathcal{A}$. The node-averaged complexity of an algorithm $\mathcal{A}$ on a family of graphs $\mathcal{G}$ is defined as follows.

$$\mathsf{AVG}_V(\mathcal{A}) := \max_{G\in\mathcal{G}} \frac{1}{|V|} \cdot \mathbb{E}\left[\sum_{v\in V(G)} T_v^G(\mathcal{A})\right] = \max_{G\in\mathcal{G}} \frac{1}{|V|} \cdot \sum_{v\in V(G)} \mathbb{E}\left[T_v^G(\mathcal{A})\right]$$

The complexity of Π is defined as the lowest complexity of all the algorithms that solve Π.

LCLs in the black-white formalism. We next define the class of problems that we consider: LCLs in the black-white formalism. A problem Π described in the black-white formalism is a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_W, C_B)$, where:

- Σ_{in} and Σ_{out} are finite sets of labels.
- C_W and C_B are both multisets of pairs, where each pair $(\ell_{\text{in}}, \ell_{\text{out}})$ is in $\Sigma_{\text{in}} \times \Sigma_{\text{out}}$.

Solving a problem Π on a graph G means that:

- $G = (W \cup B, E)$ is a graph that is properly 2-colored, and in particular each node $v \in W$ is labeled $c(v) = W$, and each node $v \in B$ is labeled $c(v) = B$.
- To each edge $e \in E$ is assigned a label $i(e) \in \Sigma_{\text{in}}$.
- The task is to assign a label $o(e) \in \Sigma_{\text{out}}$ to each edge $e \in E$ such that, for each node $v \in W$ (resp. $v \in B$) it holds that the multiset of incident input-output pairs is in C_W (resp. in C_B).

Note that when expressing a given LCL problem on a tree T in the black-white formalism, we often have to modify the tree T as follows. We subdivide every edge e of T by inserting one node in the middle of the edge. Each edge is then split into two "half-edges" and the new tree is trivially properly 2-colored (say the original nodes of T are the black nodes and the newly inserted nodes for each edge of T are the white nodes). In the full version [1], we prove that on trees, for any standard LCL, we can define an LCL in the black-white formalism that has the same asymptotic node-averaged complexity as the original one, implying that our results hold for all standard LCLs as well.

2.1 A Generic Way to Solve All LCLs

The content of this section is heavily based on results presented in [13, 10, 4]. We define two elementary procedures.

- **Rake:** Every node of degree 1 or 0 gets removed.
- **Compress(b):** Every path of degree-2 nodes of length at least b is split into subpaths of length in $[\ell, 2\ell]$ by ignoring one node between each two such subpaths. We remove all of the subpaths and leave the ignored nodes to be removed by the next rake step.

We first start by providing an $O(D)$ rounds algorithm: iteratively apply rake until the entire tree is removed; at each step, each node v that becomes a leaf computes the set L of all labels that can be put on the edge connecting v to its parent, satisfying that, for any choice in L, it is possible to pick labels from the sets assigned to the edges connecting v to its children, in a valid manner; once all edges get a set assigned, it is possible to pick a valid labeling for all the edges by processing nodes in reverse order.

Informally, we call the set of labels computed by a node the *class* of the subtree rooted at that node. In order to obtain algorithms that are faster than $O(D)$ rounds, we must also handle nodes of degree 2, and hence we also have to take care of paths obtained with the compress operation. Observe that each such path has *two* parents, that is, the two nodes of higher layers connected to its endpoints. We hence need a way to assign a class to the whole path, as a function of the classes of the subtrees of lower layers connected to the nodes of the path. However, this is challenging: how do we even *define* the class of a path? We would like that, when we process nodes in reverse order, no matter what are the *two* labels that get chosen for the edges connecting the endpoints of the path to nodes of higher layers, we can still complete the labeling inside the path (and in the subtrees connected to it). Hence, we still want to assign a set of labels to each edge connecting the path to its parents, but now these two sets must satisfy some sort of independence property. We call a pair of sets of labels that satisfy this property an *independent class* of the path.

In [13], Chang and Pettie showed that for every LCL that is solvable in $O(\log n)$ rounds, there exists some constant ℓ such that, if the paths have length at least ℓ, then there is always a way to compute an independent class, such that the algorithm sketched above works. In [10] it is then shown that a similar procedure works for all problems with complexity $\Theta(n^{1/k})$. The actual complexity of a problem is determined by the number of compress layers for which it is possible to compute an independent class. Let k be this number. If $k = \infty$, the problem Π has worst-case complexity $O(\log n)$, while if k is some constant then Π has worst case complexity $\Theta(n^{1/k})$. We will now see how to decompose any tree such that we apply the compress operation only k times.

Tree decompositions. All problems with worst-case complexity $O(\log n)$ or $O(n^{1/k})$ for any $k \in \mathbb{N}$ can be solved by following a generic algorithm [13, 10, 4]. This algorithm decomposes the tree into layers by iteratively removing nodes in a rake-and-compress manner [21] (and then uses the computed decomposition to solve the given problem). We first define the decomposition and then elaborate on how fast (and how) it can be computed.

▶ **Definition 4** ((γ, ℓ, L)-decomposition)**.** *Given three integers γ, ℓ, L, a (γ, ℓ, L)-*decomposition *is a partition of $V(G)$ into $2L-1$ layers $V_1^R = (V_{1,1}^R, \ldots, V_{1,\gamma}^R), \ldots, V_L^R = (V_{L,1}^R, \ldots, V_{L,\gamma}^R)$, $V_1^C, \ldots, V_{L-1}^C$ such that the following hold.*

1. *Compress layers: The connected components of each $G[V_i^C]$ are paths of length in $[\ell, 2\ell]$, the endpoints have exactly one neighbor in a higher layer, and all other nodes do not have any neighbor in a higher layer.*

2. *Rake layers: The diameter of the connected components in $G[V_i^R]$ is $O(\gamma)$, and for each connected component at most one node has a neighbor in a higher layer.*
3. *The connected components of each sublayer $G[V_{i,j}^R]$ consist of isolated nodes. Each node in a sublayer $V_{i,j}^R$ has at most one neighbor in a higher layer or sublayer.*

In [13, 10] it is shown how to compute a (γ, ℓ, L)-decomposition where, at the end, each node knows the layer that it belongs to. On a high level, the algorithm alternates between performing γ Rake operations and one Compress(ℓ) operation, until the empty tree is obtained. The following lemma provides upper bounds for the deterministic worst-case complexity of computing a (γ, ℓ, L)-decomposition using the algorithm of [13, 10].

▶ **Lemma 5** ([13, 10]). *Assume $\ell = O(1)$. Then the following hold.*

- *For any positive integer k and $\gamma = n^{1/k}(\ell/2)^{1-1/k}$, a (γ, ℓ, k)-decomposition can be computed in $O(kn^{1/k})$ rounds.*
- *For $\gamma = 1$ and $L = O(\log n)$, a (γ, ℓ, L)-decomposition can be computed in $O(\log n)$ rounds.*

By Lemma 5, we get that if in the former case we set $L = k$, and in the latter case we set $L = O(\log n)$, then a (γ, ℓ, L)-decomposition can be computed within a running time that matches the target complexity. In [13, 10] it is shown how to determine the value of ℓ in each case. We now define a total order on the layers of a (γ, ℓ, L)-decomposition in the natural way. This will be useful in the design of our algorithm in Section 3.

▶ **Definition 6** (layer ordering). *We define the following total order on the (sub)layers of a (γ, ℓ, L)-decomposition.*

- $V_{i,j}^R < V_{i',j'}^R$ *iff* $i < i' \vee (i = i' \wedge j < j')$
- $V_{i,j}^R < V_i^C$
- $V_i^C < V_{i+1,j}^R$

Accordingly, we will use terms such as "lower layer" to refer to a layer that appears earlier in the total order than some considered other layer.

The generic algorithm with optimal worst-case complexity. We now explain the algorithm due to [13, 10, 4] that is able to solve any LCL Π with worst-case complexity $\Theta(\log n)$ or $\Theta(n^{1/k})$ asymptotically optimally.

1. Determine ℓ and k from the description of Π. Compute γ, L accordingly.
2. Compute a (γ, ℓ, L)-decomposition, while also propagating label sets up.
3. Any node without neighbours in a higher layer picks a solution based on the label sets of their children and propagate their choice downwards.
4. Nodes that receive a choice from their parents simply pick a label respecting the label sets of their children.

Because of the way the label sets were chosen all nodes will be able to pick a valid label. For the details of how the label sets are chosen and why all nodes will be able to pick a valid label, we refer to the full version of the paper [1].

We note that the generic algorithm does not require a specific (γ, ℓ, L)-decomposition – any (γ, ℓ, L)-decomposition (for the parameters γ, ℓ, L determined in the beginning of the generic algorithm) works. We will make use of this fact when designing algorithms with a good node-averaged complexity in Section 3.

3 Algorithm for Intermediate Worst-Case Complexity Problems

In this section, we provide an algorithm with node-averaged complexity $O(\log^* n)$ on bounded-degree trees for all LCLs that can be solved in worst-case complexity $O(\log n)$ on such trees. The main difference to the generic algorithm is that we compute our (γ, ℓ, L)-decomposition in such a way that we obtain many more local maxima, that is, nodes with a layer number that is higher than the ones of all its neighbors. We achieve this by leaving slack at the end of compress paths and then inserting new compress paths to create local maxima.

3.1 The Decomposition Algorithm

Our algorithm is essentially a modified rake and compress procedure. During the execution of the algorithm we will maintain a partial (γ, ℓ, L)-decomposition and change it to create local maxima where we want them. For now we start by distinguishing between nodes that have already been assigned a layer and those that have not.

▶ **Definition 7** (free and assigned nodes). *We call a node that has not been assigned to a layer a* free *node. A node that has been assigned to a layer is called* assigned.

Let G denote our input tree and assume that a subset of nodes has already been assigned to some layers. Let G' denote the subgraph of G induced by all assigned nodes. Recall that we have a total ordering on the layers of a decomposition due to Definition 6 (that naturally extends to partial decompositions). Our aim is to create local maxima where they are most useful to us.

▶ **Definition 8** (local maximum). *A* local maximum *is an assigned node $v \in V(G')$ with the following two properties:*

1. *Node v and all of its neighbors are assigned, i.e., they are all contained in $V(G')$.*
2. *For each neighbor w of v, the layer of w is strictly smaller than the layer of v.*

In our algorithm, we will artificially promote some nodes to a higher layer in order to produce local maxima. We choose which node to promote according to the *quality* of the nodes. The quality of a node v is the number of nodes that are waiting for v to propagate its label downwards. So if v terminates and chooses an output, then all of these nodes can also terminate. To keep track of these dependencies, we orient (some of the) edges of the input graph in such a way that any node u will have an oriented path from v to u if and only if u will be able to terminate if v does (see Figure 2). In practice, this means that, if u is raked away, we orient the single edge from u's parent towards u and we orient the ends of compress paths inwards. If an edge is not explicitly oriented it is not considered oriented at all. For some given v this orientation now defines $H(v)$, a subgraph of nodes that can be reached over oriented paths. In other words, $H(v)$ contains all the nodes that are only waiting for v to choose an output and hence these could all terminate if v became a local maximum.

▶ **Definition 9** (quality). *For any node $v \in V(G)$, let $H(v)$ denote the set of all nodes w that can be reached from v via a path $(v = v_0, v_1, \ldots, v_j = w)$ such that the following hold:*

1. *The edge $\{v_{i-1}, v_i\}$ is oriented from v_{i-1} to v_i, for each $1 \leq i \leq j$.*
2. *All nodes on the subpath from v_1 to w are assigned, i.e., they are all contained in $V(G')$.*
3. *The layer of v_i is smaller than or equal to the layer of v_{i-1}, for each $2 \leq i \leq j$, and if v_0 is assigned, then the layer of v_1 is smaller than or equal to the layer of v_0.*

If v is a local maximum, or a descendant of a local maximum, then the quality $q(v) := 0$. Otherwise the quality *$q(v)$ of a node v is the number of nodes in $H(v)$, i.e., $q(v) := |H(v)|$.*

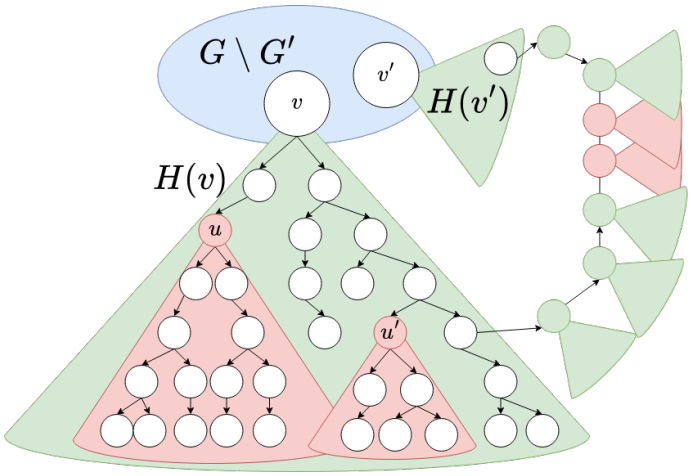

Figure 2 v and v' are free nodes. $H(v)$ (respectively, $H(v')$) contains all nodes inside the green cone attached to v (respectively, v'). The path connecting $H(v)$ and $H(v')$ is a compress path were the ends are oriented inwards. As a result the green nodes at the end of that path contribute to $H(v)$ and $H(v')$ respectively. u and u' are local maxima, so because of Item 3 in Definition 9 they and the red trees hanging from them do not contribute to $H(v)$ (respectively, $H(v')$).

We chose the name quality, since we will later decide on which nodes to fix by trying to maximize this quantity. To be more precise when stating the algorithm we also introduce the following notions based on the orientation.

▶ **Definition 10** (child, parent, descendant, ancestor, orphan)**.** *For any edge $\{w, w'\}$ oriented from w to w', we call w' a* child *of w and w the parent of w'. For any oriented path $(w, \ldots, w')$ that is consistently oriented from w to w', we call w' a* descendant *of w and w an* ancestor *of w'. We call a node with no edges oriented towards itself an* orphan*.*

In previous work the rake and compress procedure is done by iteratively performing some rakes and then one compress. We change the ordering by first performing one compress and then some rakes. To still get the same guarantees our algorithm initially performs γ rakes. Each compress operation is done in a modified, non-standard, way: normally to remove as many nodes as possible in each iteration, we would like to compress paths that are as short as possible. Instead we make sure we have some extra slack at the end of each compress path. This is to ensure that compress paths are always far enough away from nodes that we want to promote. We then rake away this slack, by performing a set of γ rakes.

After we are done with compress and rakes we want to promote some node v^*. We define the set $C_b(r)$ as the set of descendants of r that have distance exactly b from r. In Figure 2, e.g., $C_2(v)$ are the nodes inside the green cone of v that are at distance exactly 2 from v, so $u \in C_2(v)$. We determine the node that we want to promote as the node v^* of highest quality among all nodes in $C_b(r)$. We will later see that if v^* is promoted, the quality of r is reduced by a constant factor. We further define $G^{(i)}$ as the set of free nodes at the end of iteration i. The full details of the algorithm are given in the full version [1]. Note that Algorithm 1 provides a description of the steps of the algorithm without specifying how the algorithm is implemented in a distributed manner. We will take care of the latter in Section 3.3.

We will use this result to show that $O(\log n)$ iterations of Algorithm 1 are enough.

▶ **Lemma 11** ([10])**.** *Given a tree with n nodes, by performing α rakes and 1 compress with minimum path length β, the number of remaining nodes is at most $\frac{\beta}{2\alpha}n$.*

Algorithm 1 Compute Decomposition *Informal*.

Input: $G = (V, E), \Pi$
compute ℓ from Π
$b \leftarrow \ell + 2$
$\gamma \leftarrow \ell + 3$
Perform γ orienting Rakes
for $O(\log n)$ *times (until every node is assigned to a layer)* **do**
  for *each path with length at least* $4\ell + 9$ **do**
    Ignore the first and last γ nodes ▷ Will be raked away.
    Perform normal compress on the truncated path
    Orient the ends inwards
  Perform γ orienting Rakes ▷ Thereby raking away the slack from compress
  for *each free node* r **do**
    v^* is the descendant at distance b with the largest quality
    if *The path from* r *to* v^* *does not intersect with any compress path* **then**
      Promote v^* into a local maximum by reassigning the path from r to v^* to a compress layer with v^* as the end point in the next higher rake layer

To show that the algorithm is correct we prove that it computes a valid (γ, ℓ, L)-decomposition by showing that it always maintains a valid partial (γ, ℓ, L)-decomposition.

▶ **Lemma 12.** *After every iteration i the partial assignment of nodes to layers at time t forms a partial $(\gamma, \ell, i+1)$-decomposition. Also at the end we get a (γ, ℓ, L)-decomposition with $L \in O(\log n)$.*

Proof. We prove the lemma by induction on the current iteration i. Everything except the promotion of a node is trivial, so for this case we show that all three properties of Definition 4 continue to hold. The nodes in the promoted path of length $b - 1$ (without v^*) are now put into a compress layer. Since the path has length $b - 1$ and does not intersect with any other compress layer, we guarantee that Item 1 still holds for all compress layers. Item 2 and Item 3 still hold since we only take away from old rake layers and hence cannot invalidate these properties. Since we satisfy Lemma 11 every iteration we need only $L \in O(\log n)$ iterations. ◀

In the next section our main goal is to prove that enough of these local maxima actually exist and are nicely distributed in the graph.

3.2 Local Maxima and Bounding Quality

We will first see that during the execution of our algorithm we can actually decompose the graph into a bunch of subtrees as seen in Figure 2. Consider iteration i and the corresponding set of still free nodes $G^{(i)}$. We now give one of the most important definitions, that of a subtree of assigned nodes (refer to Figure 2 to get an intuition).

▶ **Definition 13** (subtree of assigned nodes)**.** *For any node $v \in V(G)$ in any iteration i, the subtree of assigned nodes $T^{(i)}(v)$ denotes the set that contains v and all nodes w that can be reached from v via a path $(v = v_0, v_1, \ldots, v_j = w)$ such that edge $\{v_{a-1}, v_a\}$ is oriented from v_{a-1} to v_a, for each $1 \leq a \leq j$. Additionally $h^{(i)}(v) = \max\{\text{dist}(v, u)\}_{u \in T^{(i)}(v)}$ is the height of the tree.*

We now express the entire input graph G in terms of these trees. Notice that any node must either be an orphan or be in the subtree of assigned nodes of some orphan. As a result the union over all trees of all orphans will cover the entire tree. We define sets $N^{(i)}$ which contain all of the orphans that are created during our compress procedures. For every path that gets compressed (i.e., not including promoted compress paths) up until the end of iteration i, we add to $N^{(i)}$ all nodes in the path except the oriented parts at the beginning and the end. The sets are chosen exactly such that all orphans are in one or the other set, so we obtain the following.

▶ **Lemma 14.** *For each positive integer i,the following holds:*

$$V(G) = \left(\bigcup_{v \in G^{(i)}} T^{(i)}(v) \right) \cup \left(\bigcup_{v \in N^{(i)}} T^{(i)}(v) \right)$$

Furthermore the two big unions are disjoint.

Notice that for each of these subtrees of assigned nodes, the quality of the root counts exactly how many nodes in this tree still need to terminate. Notice however that all nodes in $N^{(i)}$ are either local maxima or between two local maxima and they can thus already terminate. So by giving a good upper bound on the quality of nodes in $G^{(i)}$, we will show that in each iteration a constant fraction of the remaining nodes terminate.

Creating Local maxima. We next narrow down our view to some concrete iteration i and will hence drop some of the (i) in the exponents. Just from the design of the algorithm we get that if the if-statement in Algorithm 1 is true and we promote v^*, v^* will indeed be a local maximum.

▶ **Lemma 15.** *After v^* is promoted, v^* will be a local maximum.*

Now we will see that if we do promote v^* to become a local maximum, we get that $q(v^*)$ will be a large part of $q(r)$, thereby showing that with each promotion we reduce the quality by a constant fraction.

▶ **Lemma 16.** *If v^* is promoted, then $q(v^*) \geq \frac{q(r)}{2\Delta^b}$.*

Proof. The statement follows from the fact that $q(r) \leq \Delta^b + \sum_{v \in C_b(r)} q(v)$. This is true, because we can separate $H(r)$ into the nodes that are close and those that are far. Concretely a node u is close, if the path to r is strictly less than b. But only Δ^b such nodes can exist. A node u is far, if the path to r is at least b nodes long, at which point it has to pass through one of the nodes $v \in C_b(r)$. Now since $u \in H(r)$ the unique path from r to u satisfies the criteria for u to be in $H(r)$, then the subpath from w to u must also satisfy these criteria and u is therefore included in $q(w)$. We then get

$$q(r) \leq \Delta^b + \sum_{v \in C_b(r)} q(v) \leq \Delta^b + |C_b(r)| \cdot q(v^*) \leq \Delta^b + \Delta^b q(v^*) \leq 2\Delta^b q(v^*).$$

As a result $u \in H(v)$ and therefore u is accounted for in $q(v)$. The second inequality comes from the fact v^* has the highest quality among nodes in $C_b(r)$. ◀

However the if-condition may not hold in every iteration, but this then means that there is a compress path within distance b from r. This compress path cannot be from the normal compress procedure, because of the γ nodes of slack we leave at the ends of every path. As a consequence this path is due to a previous promotion; let x be that promoted node. This allows us to prove the following.

▶ **Lemma 17.** *If the if-condition does not hold, then there exists a promoted node x at distance at most $2b-1$ from r, such that $q(x) \geq \frac{q(v^*)}{2\Delta^b}$ immediately before x was promoted.*

Upperbounding the quality of a free node. The next lemma will be the main technical result that shows that enough nodes are in subtrees of local maxima. We will show this implicitly by upperbounding the quality of the remaining free nodes. However we will have to introduce some more notation. We are partitioning each assigned subtree $T^{(i)}(v)$ into $\alpha = \left\lceil \frac{h^{(i)}(v)+1}{b} \right\rceil$ subsets $S_0^{(i)}(v), \ldots, S_{\alpha-1}^{(i)}(v)$, where, for each $0 \leq j \leq \alpha - 1$,

$$S_j^{(i)}(v) := \{u \in T^{(i)}(v) \mid b \cdot j \leq \mathrm{dist}(u, v) < b \cdot (j+1)\}.$$

Note that we have $\bigcup_{0 \leq j \leq \alpha-1} S_j^{(i)}(v) = T^{(i)}(v)$.

▶ **Lemma 18.** *There exists a constant $0 < \lambda < 1$ (that only depends on Π and Δ) such that for all iterations i, the following inequality holds at the end of iteration i, for all nodes $r \in G^{(i)}$:*

$$q^{(i)}(r) \leq \sum_{j=0}^{\lceil (h^{(i)}(r)+1)/b \rceil - 1} \lambda^j |S_j^i(r)|$$

Intuitively, the deeper in the subtree the nodes are, the more of them are already fixed. Since most of the nodes get removed in the first few iterations, this suffices. The full proof can be found in the full version [1]. Glossing over a lot of details, the main idea is to use induction over the height of the tree and use the induction hypothesis on all of the nodes in $C_b(r)$ to obtain an initial bound on the quality of r. Then we note that the promotion of v^* is not yet taken into account in this bound and we get the desired result. If v^* cannot be promoted, by Lemma 17 some node x was recently promoted. A careful analysis shows that also x was not yet taken into account in the initial bound and we get the desired results.

3.3 Distributed Algorithm and Node Averaged Complexity

In this section, we will describe how we implement Algorithm 1 in a distributed manner and how we will use it to design an algorithm $\mathcal{A}$ that solves the given LCL problem Π, and we will prove an upper bound of $O(\log^* n)$ for the node-averaged complexity of the latter algorithm. We start by describing our distributed implementation of Algorithm 1. For the remainder of the section, set $s := 10\ell$.

Distributed implementation. The computation of ℓ from Π can be performed by every node without any communication. Next, the nodes compute a distance-s coloring with a constant number of colors. Since Δ and ℓ are constant, this can be done in worst-case complexity $O(\log^* n)$, e.g., by using an algorithm of [8]. Using this coloring we can execute compress operations in a constant number of rounds, by simply iterating through the color classes. The γ rakes can trivially be implemented in γ rounds. the promotion requires only to see up to a constant distance, as long as the qualities of all nodes are known. However these can be computed on the fly during the algorithm.

▶ **Lemma 19.** *Assume a distance-s coloring with a constant number of colors is given. Then iteration i of Algorithm 1 can be executed in a constant number of rounds, for each even positive integer i.*

Next we describe our algorithm $\mathcal{A}$ for solving a given LCL problem Π.

Algorithm for Π. Algorithm $\mathcal{A}$ proceeds as follows. Use Algorithm 1 to compute a (γ, ℓ, L)-decomposition, where the values of γ and ℓ depend on Π, and $L \in O(\log n)$ (due to Lemma 12). As soon as a node becomes a local maximum it starts to execute the steps from the original generic algorithm to compute its class, pick an output and start propagating downwards. As a result the entire subtree of assigned nodes of such a local maximum will terminate. The only thing needed for this are the label sets, but they can be propagated upwards during the execution of the decomposition algorithm with no additional cost. As a result we obtain the following lemma.

▶ **Lemma 20.** *Assume a distance-s coloring with a constant number of colors is given. Then there exists an integer constant t such that the following holds: if a node v becomes a local maximum in iteration i of Algorithm 1, then the entire tree $T^{(i)}(v)$ will have terminated after ti rounds in $\mathcal{A}$.*

To make the runtime analysis a bit cleaner, we are going to mark all nodes in $T^{(i)}(v)$, once v becomes a local maximum. We emphasize that this is solely for the purpose of the analysis and this does not change the algorithm at all. More specifically, once any node v becomes a local maximum, all of the nodes in $T^{(i)}(v)$ become marked instantly (in 0 rounds). We obtain the following corollary from Lemma 20.

▶ **Corollary 21.** *Assume a distance-s coloring with a constant number of colors is given. If a node v becomes marked in iteration i, then v will have terminated in round ti, where t is the constant from Lemma 20.*

Consider some iteration i, by applying Lemma 18 on every remaining free node in $G^{(i)}$ we can upperbound the total quality and therefore the total number of unmarked nodes. For the full proof refer to the full version of the paper [1].

▶ **Lemma 22.** *There exists a constant $0 < \sigma < 1$, such that for every iteration $i \geq 10$ of Algorithm 1 at most $2\Delta^b n\sigma^i$ nodes are not marked.*

We obtain the following lemma.

▶ **Lemma 23.** *On average, nodes become marked in $O(1)$ iterations.*

Then using this lemma together with Corollary 21 we get that an average node terminates after a constant number of rounds. However, we still have to pay for the input distance coloring which takes $O(\log^* n)$, as discussed in the beginning of the section. So by first computing this input coloring and then running the algorithm, we obtain a total node-averaged complexity of $O(\log^* n)$, proving Theorem 1.

References

1 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, Dennis Olivetti, and Gustav Schmid. On the node-averaged complexity of locally checkable problems on trees. *CoRR*, abs/2308.04251, 2023. doi:10.48550/arXiv.2308.04251.

2 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*, pages 299–308. ACM Press, 2020. doi:10.1145/3382734.3405715.

3 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. *Distributed Comput.*, 34(4):259–281, 2021. doi:10.1007/s00446-020-00375-2.

4 Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally checkable labelings with small messages. In *35th International Symposium on Distributed Computing, DISC 2021*, pages 8:1–8:18, 2021. doi:10.4230/LIPIcs.DISC.2021.8.

5 Alkida Balliu, Mohsen Ghaffari, Fabian Kuhn, and Dennis Olivetti. Node and edge averaged complexities of local graph problems. In *Proc. 41st ACM Symp. on Principles of Distributed Computing (PODC)*, pages 4–14, 2022.

6 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC 2018)*, pages 1307–1318. ACM Press, 2018. doi:10.1145/3188745.3188860.

7 Alkida Balliu, Juho Hirvonen, Dennis Olivetti, and Jukka Suomela. Hardness of minimal symmetry breaking in distributed computing. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 369–378. ACM Press, 2019. doi:10.1145/3293611.3331605.

8 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. *SIAM J. on Computing*, 43(1):72–95, 2015.

9 Leonid Barenboim and Yaniv Tzur. Distributed symmetry-breaking with improved vertex-averaged complexity. In *Proc. 20th Int. Conf. on Distributed Computing and Networking (ICDCN)*, pages 31–40, 2019.

10 Yi-Jun Chang. The complexity landscape of distributed locally checkable problems on trees. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.18.

11 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. Distributed edge coloring and a special case of the constructive lovász local lemma. *ACM Trans. Algorithms*, 16(1):8:1–8:51, 2020.

12 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. *SIAM J. Comput.*, 48(1):122–143, 2019. doi:10.1137/17M1117537.

13 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

14 Soumyottam Chatterjee, Robert Gmyr, and Gopal Pandurangan. Sleeping is efficient: MIS in $O(1)$-rounds node-averaged awake complexity. In *Proc. 39th ACM Symp. on on Principles of Distributed Computing (PODC)*, pages 99–108, 2020.

15 Laurent Feuilloley. How long it takes for an ordinary node with an ordinary ID to output? In *Proc. 24th Int. Coll. on Structural Information and Communication Complexity (SIROCCO)*, volume 10641, pages 263–282, 2017.

16 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *LIPIcs*, pages 18:1–18:16, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

17 Christoph Grunau, Václav Rozhon, and Sebastian Brandt. The landscape of distributed complexities on trees and beyond. In *Proc. 41st ACM Symposium on Principles of Distributed Computing (PODC 2022)*, pages 37–47, 2022. doi:10.1145/3519270.3538452.

18 Öjvind Johansson. Simple distributed *Delta*+1-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232, 1999.

19 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, 2016.

20 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

21 Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 478–489. IEEE Computer Society, 1985. doi:10.1109/SFCS.1985.43.

22 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

23 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proc. 52nd ACM Symp. on Theory of Computing (STOC)*, pages 350–363, 2020.

A Improved Algorithms in the Polynomial Regime

In this section we show that, for infinitely many LCL problems with polynomial worst-case complexity, we can improve their node-averaged complexity. More precisely, in this section we show that, for a class of problems with worst-case complexity $\Theta(n^{1/k})$, we can provide an algorithm with node-averaged complexity $O(n^{1/(2^k-1)})$. As we show in Appendix B, this complexity is almost tight, since for all problems with worst-case complexity $\Theta(n^{1/k})$ we can show a lower bound of $\widetilde{\Omega}(n^{1/(2^k-1)})$.

The Hierarchical $2\frac{1}{2}$-Coloring Problems

We now define a class of problems, already presented in [13], called hierarchical $2\frac{1}{2}$-coloring, that is parametrized by an integer $k \in \mathbb{Z}^+$. It has been shown in [13] that the problem with parameter k has worst-case complexity $\Theta(n^{1/k})$. We now give a formal definition of this class of problems.

The set of input labels is $\Sigma_{\text{in}} = \emptyset$. The set of output labels contains four possible labels, that is, $\Sigma_{\text{out}} = \{W, B, E, D\}$, and these labels stand for *white*, *black*, *exempt*, and *decline*. Each node has a level in $\{1, \dots, k+1\}$, that can be computed in constant time, and the constraints of the nodes depend on the level that they have. The level of a node is computed as follows.

1. Let $i \leftarrow 1$.
2. Let V_i be the set of nodes of degree at most 2 in the remaining tree. Nodes in V_i are of level i. Nodes in V_i are removed from the tree.
3. Let $i \leftarrow i + 1$. If $i \leq k$, continue from step 2.
4. Remaining nodes are of level $k + 1$.

Each node must output a single label in Σ_{out}, and based on their level, they must satisfy the following local constraints.

- No node of level 1 can be labeled E.
- All nodes of level $k + 1$ must be labeled E.
- Any node of level $2 \leq i \leq k$ is labeled E iff it is adjacent to a lower level node labeled W, B, or E.
- Any node of level $1 \leq i \leq k$ that is labeled W (resp. B) has no neighbors of level i labeled B (resp. W) or D. In other words, W and B are colors, and nodes of the same color cannot be neighbors in the same level.
- Nodes of level k cannot be labeled D.

This problem can be expressed as a standard LCL by setting the checkability radius r to be $O(k)$, since in $O(k)$ rounds a node can determine its level and hence which constraints apply. In Section 1.2 we provided some intuition about these problems. We provide more intuition in the full version of the paper [1].

Better Node-Averaged Complexity

We now show that, for this class of LCL problems, we can obtain a better node-averaged complexity. The algorithm is similar to the one presented in [10] for the worst-case complexity, but it is modified to obtain a better node-averaged complexity (in Appendix B we show that this algorithm is tight up to a $\log n$ factor).

▶ **Theorem 24.** *The node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$.*

Proof. At first, all nodes spend $O(1)$ rounds to compute their level. Nodes of level $k+1$ output E. Then, the algorithm proceeds in phases, for i in $1,\dots,k$. In phase i, all nodes of level i get a label, and hence let us assume that all nodes of levels $1,\dots,i-1$ already have a label, and let us focus on level-i nodes.

Consider a node v of level i. Node v proceeds as follows. If v has a neighbor from lower levels that is labeled W or B then v outputs E. Otherwise, v spends $t_i = c \cdot \gamma_i$ rounds to check the length of the path containing v induced by nodes of level i, for some constant c to be fixed later, and $\gamma_i = n^{2^{i-1}/(2^k-1)}$. If this length is strictly larger than t_i, then v outputs D. Otherwise, all nodes of the path are able to see the whole path, and hence they can output a consistent 2-coloring by using the labels W and B.

The above algorithm correctly solves the problem if we assume that no nodes in level k output D. In the full version of the paper [1] we show that indeed nodes of level k do not output D, hence showing the correctness of the algorithm, and then we prove a bound on the node-averaged complexity. In the following we sketch the main idea for the correctness of the algorithm.

Let S be the set of nodes of level i that do not directly output E at the beginning of phase i. It is possible to assign each node of level $j < i$ to exactly one node in S such that to each node in S are assigned $\Omega(n^{(2^{i-1}-1)/(2^k-1)})$ unique nodes of lower layers. Hence, the number of nodes that participate in phase i is at most $O(n^{1-(2^{i-1}-1)/(2^k-1)}) = O(n^{(2^k-2^{i-1})/(2^k-1)})$. This implies that in phase k the number of participating nodes is at most $O(n^{(2^k-2^{k-1})/(2^k-1)}) = O(n^{2^{k-1}/(2^k-1)}) = O(\gamma_k)$, where the hidden constant is inversely proportional to c. Hence, by picking c large enough, we get that in t_k rounds nodes of level k see the whole path and thus no node of level k outputs D, proving the correctness of the algorithm. ◀

B Lower Bounds in the Polynomial Regime

In this section we show that any LCL problem that requires polynomial time for worst-case complexity requires polynomial time also for node-averaged complexity. More precisely, we prove the following theorem.

▶ **Theorem 25.** *Let Π be an LCL problem with worst-case complexity $\Omega(n^{1/k})$ in the* LOCAL *model. The node-averaged complexity of Π in the* LOCAL *model is $\Omega(n^{1/(2^k-1)}/\log n)$.*

In order to prove this theorem, we proceed as follows (throughout this section we will use notions presented in Section 2.1). It is known by [10] that if an LCL problem Π has worst-case complexity $o(n^{1/k})$, then it can actually be solved in $O(n^{1/(k+1)})$ rounds. This statement is proved by showing that it is possible to use an algorithm (possibly randomized) running in $o(n^{1/k})$ rounds to construct a function $f_{\Pi,k+1}$ that can be used to map the label sets assigned to the edges connected to a compress path from lower layers into two label sets for the edges connecting the endpoints of the path to their parents, in such a way

that, if the compress layers are at most $k+1$, then the algorithm sketched in Section 2.1 works. This implies, as shown in Section 2.1, the existence of a deterministic algorithm that solves Π and has worst-case complexity $O(n^{1/(k+1)})$. In this section we show that it is possible to construct a function $f_{\Pi,k+1}$ by starting from an algorithm $\mathcal{A}$ with node-averaged complexity $o(n^{1/(2^k-1)}/\log n)$. By Section 2.1, this implies that if there exists an algorithm with $o(n^{1/(2^k-1)}/\log n)$ node-averaged complexity, then there exists an algorithm with worst-case complexity $O(n^{1/(k+1)})$, implying that any LCL with worst-case complexity $\Omega(n^{1/k})$ has node-averaged complexity at least $\Omega(n^{1/(2^k-1)}/\log n)$. While we will use some ideas already presented in [10], handling an algorithm with only guarantees on its node-averaged complexity arises many (new) issues that we need to tackle.

Our statement will be proved even for the case in which algorithm $\mathcal{A}$ satisfies the weakest possible assumptions (i.e., the assumptions are so relaxed that they are satisfied by any deterministic algorithm, any randomized Las Vegas algorithm, and any randomized Monte Carlo algorithm). The assumptions are the following.

- We assume that $\mathcal{A}$ is a randomized algorithm that is only required to work when the unique IDs of nodes are assigned at random, among all possible valid assignments.
- We assume that $\mathcal{A}$ is an algorithm that fails with probability at most $1/n^c$ for any chosen constant $c \geq 1$.
- We assume that the bound on the node-averaged complexity of $\mathcal{A}$ holds with probability at least $1 - 1/n^c$ for any chosen constant $c \geq 1$.

However, in the following, we will assume that the bound on the node-averaged complexity of $\mathcal{A}$ holds always. In fact, observe that we can always convert an algorithm with node-averaged complexity T that holds with probability at least $1 - 1/n^c$ into an algorithm with node-averaged complexity $O(T)$ that holds always, since, even for $c = 1$, we can safely assume that when the bound does not hold (that happens with probability at most $1/n$), the runtime is anyways bounded by n (since everything can be solved in n rounds in the LOCAL model).

Proof Overview

We now sketch the high-level ideas for proving that it is possible to use an algorithm with $o(n^{1/(2^k-1)}/\log n)$ node-averaged complexity to construct a function $f_{\Pi,k+1}$. All the details are deferred to the full version of the paper [1], where, for completeness, we also prove a result already presented in [13, 10]: Whether a function $f_{\Pi,k+1}$ exists can be mechanically determined.

Recall that the input of the function $f_{\Pi,k+1}$ is a path, where to each node are connected some edges (that we call incoming edges), and for each of them is provided a set of labels. Additionally, to each endpoint of the path is connected an additional edge (called outgoing edge). We need to compute two sets of labels, L_1 and L_2, one for each outgoing edge. These sets must satisfy that, for any choice of labels $(\ell_1, \ell_2) \in L_1 \times L_2$, we can pick a label from the sets assigned to the incoming edges, and a label for each edge of the path, that results in a labeling that satisfies the constraints of the problem. Observe that there may exist multiple pairs of non-empty sets satisfying the requirements, but we need a specific type of solution: once such sets are propagated to higher layers, we still want to be able to perform the same operation in the next compress layer. A more general property that needs to be satisfied is that empty sets are never obtained, because this would prevent nodes from being able to pick a valid labeling in the top-down phase. In [13, 10] it is shown how to construct such a function, by starting from an algorithm with (possibly randomized) worst-case complexity $o(n^{1/k})$. On a high level, this is achieved as follows:

1. Replace each incoming edge of the path with a small tree, satisfying that the class of the tree corresponds to the label set of the edge.
2. Modify the path, by making it *much longer*, but by preserving its completability properties, that is, a choice $(\ell_1, \ell_2) \in L_1 \times L_2$ is good for the long version of the path if and only if it is good for the original version.
3. Ask the algorithm what it would output in the middle of such a long path. Crucially, the path is made so long that the algorithm, within its running time, cannot see the endpoints.
4. Compute what labels can be put on the outgoing edges of the long path, in a way that the labeling in the rest of the path can be completed in a valid way (and it is compatible with the label sets of the incoming edges), and such that the output in the middle of the path corresponds to the output of the algorithm. Since the output in the middle of the path is fixed, then the outputs on the outgoing edges can be chosen *independently*. Hence, in this way, we obtain an independent class for the path.

In [13, 10] it is argued why, by using such a function with the algorithm sketched in Section 2.1, empty label sets are never obtained. While we need to adapt such a proof to the case of node-averaged complexity, there are some additional challenges that we need to tackle. For example, one issue that we have when trying to adapt this approach to the case of node-averaged complexity is that an algorithm could run for a long time at *some nodes*, while still keeping a low node-averaged complexity. Hence, we do not have the guarantee that, if we make a path long, then the algorithm does not see the endpoints. In the full version of the paper we show that, in the instances that we need to handle in order to construct a valid function, we can prove that a stronger notion of node-averaged complexity must hold, namely that the expected running time of each node is bounded. Then, by considering $\Theta(\log n)$ sufficiently separated nodes of a long path, we can prove that, if we consider a node in the middle of the path and an endpoint, with high probability they are not able to communicate, and we hence obtain a result similar to the case of worst-case complexity. Moreover, the bound on the expected runtime will depend on the layer number, and this bound is what governs the final lower bound complexity.